#FIVE : High-Level Components for Developing Collaborative and Interactive Virtual Environments

Rozenn Bouville, Valérie Gouranton, Thomas Boggini, Florian Nouviale, Bruno Arnaldi

#FIVE : High-Level Components for Developing Collaborative and Interactive Virtual Environments

Rozenn Bouville * Valérie Gouranton† Thomas Boggini‡ Florian Nouviale§ Bruno Arnaldi¶

INSA de Rennes - IRISA / INRIA

ABSTRACT

This paper presents #FIVE (Framework for Interactive Virtual Environments), a framework for the development of interactive and collaborative virtual environments. #FIVE has been developed to answer the need for an easier and a faster conception and development of virtual reality applications. It has been designed with a constant focus on re-usability with as few hypothesis as possible on the final application in which it could be used. Whatever the chosen implementation for the Virtual Environment (VE), #FIVE : (1) provides a toolkit that eases the declaration of possible actions and behaviours of objects in the VE, (2) provides a toolkit that facilitates the setting and the management of collaborative interactions in a VE, (3) is compliant with distribution of the VE on different set-ups and (4) proposes guidelines to efficiently create a collaborative and interactive VE. It is composed of several modules, among them, two core modules : the **relation engine** and the **collaborative interaction engine**. On the one hand, the *relation engine* manages the relations between the objects of the environment. On the other hand, the *collaborative interaction engine* manages how users can collaboratively control objects. The modules that compose the #FIVE framework can be used either independently or simultaneously, depending on the requirements of the application. They can also communicate and work with other modules thanks to an API. For instance, a scenario engine can be plugged to any or both of the #FIVE modules if the application is scenario-based. #FIVE is a work in progress, new core modules will later be proposed. Nevertheless, it has already been used in some VR applications by several persons in our lab. The feedbacks we obtained are rather positive and we intent to further develop #FIVE with additional functionalities, notably by extending it to the control of avatars whether they are controlled by a user or by the system.

*e-mail: rozenn.bouville_berthelot@irisa.fr

†e-mail:valerie.gouranton@irisa.fr

‡e-mail:thomas.boggini@irisa.fr

§e-mail:florian.nouviale@irisa.fr

¶e-mail:bruno.arnaldi@irisa.fr

1 INTRODUCTION

Over the past 25 years, many software systems and tools have been proposed to support the building of virtual reality applications. Since the RB2 system [3], these initiatives have followed the trend to match hardware and network capacities. For that reason, but also whether the solution was more or less generic, some have met a large adoption whereas others have now been forgotten. Despite all of these initiatives, in many cases, people still prefer to build than to reuse. To our point of view, a strong impediment of many of the previous solutions is that they depend on a specific engine or on a third-party library. Indeed, when a solution is tool-dependent, it significantly hampers its usage. For instance, the tool on which the solution depends on could no longer be maintained. Another reason can be that the developer already uses a concurrent tool which he/she prefers to keep. The third-party tool could also become less efficient given hardware and/or network evolutions. These observations led us to design a solution that is totally independent to any third-party tool or engine. Thus, the #FIVE framework is an high-level toolkit that provides an abstraction layer to support the process of making a virtual environment collaborative and interactive.

The purpose of #FIVE is to support the development of our collaborative and interactive demonstrators and lower the cost of creating new ones. It focuses on how to define the behaviours of virtual objects and also on interaction in VE, including collaborative interactions. Indeed, one of the main usage of a VE is to immerse users so that they can act in the virtual world. Nevertheless, our work does not focus on AI and Intelligent Virtual Environment such as those described in [11]. To do so, every possible action and behaviour of interactive objects must be described. This mandatory task is, without an appropriate tool, fastidious and full of redundancy. The *relation engine* of #FIVE proposes a set of models to support this task; its role is to determine how and which objects of the world can be used. Furthermore, thanks to the broadening of high bandwidth networks and to the possibility of running 3D graphics on a large panel of devices, collaboration in virtual environments has become widely used. Therefore, providing models to ease the implementation of collaborative manipulation and distribution of virtual reality applications is relevant. This is the purpose of the *collaborative interaction engine* of #FIVE which provides a set of models for the collaborative control of virtual objects. The goal of the *collaborative interaction engine* is not to describe the semantic of interactions but to abstract the exchange of data for collaborative interaction without preventing the developer from de-

ciding how concurrent commands are merged and applied to the object. Eventually, the #FIVE framework comes with guidelines that allows the developer to efficiently use the tools provided by #FIVE. Indeed, as #FIVE does not made any assumption on the implementation choice of the developer, its usage is versatile.

2 Related work

Designing and implementing virtual environment (CVE) has long been recognize as a complex and a time-consuming task, especially when they are collaborative. To assist and facilitate the development of these applications, many authors agreed in the fact that the community needs tools [9]. In this section, we present some existing software systems and tools that help in the design and implementation of Collaborative Virtual Environments (CVE). We then present a selection of works that highlights guidelines for designing and implementing VE in general. Finally, we conclude this section with a discussion on requirements for frameworks for CVE.

2.1 Frameworks for CVE

In this section, we focus on frameworks that allow collaborative interaction (i.e simultaneous interaction of several users on the same object) for two reasons. First, this is a strong requirement for our own projects. Second, even among widely used and interesting frameworks like MR toolkit [15], FlowVR [1] or VRJuggler [2], it is a very restrictive criterion. Indeed, collaborative interaction is not a mandatory feature of VE but, nonetheless, it is one of the most complex feature to implement because it takes into account, at once, graphics hardware capacities and networks constraints. In addition to the collaboration criteria, we study the existing systems through 4 other criteria :

1. interaction design model, i.e the interaction level allowed by the system (device independence, definition of relations between objects, description of object behaviours, etc.),
2. distribution, i.e whether the system takes into account simultaneous executions and deployment among different platforms,
3. independence, i.e whether the system is independent from third-party tools,
4. versatility, i.e how much freedom of implementation is offered by the system,
5. factorization, i.e the level of re-usability achieved by the system.

One of the first software system to propose collaborative interactions is DIVE [5]. The DIVE system is more than 20 years old and since its first version, it went through several iterations. These iterations improved the system by adding features while preserving its independence on third-party tools. DIVE succeed in providing versatility in its usage but not in achieving re-usability.

The MASCARET system [4] is another framework for CVE. It focuses on CVE for training, nonetheless, it has already been used for other kind of application. Here, the interaction design model relies on a language to describe behaviours of objects and their relations, allowing an high level of interaction. Moreover, it also takes into account, the semantic of the interaction using ontologies. Even though MASCARET enables multi-user application we could not find information on its distribution capability. Moreover, the modules that compose MASCARET are interdependent hence a little versatility.

Similarly to MASCARET, the GVT [8] framework focuses on CVE for training. GVT provides an high interaction level through a description language that allows to precisely described behaviours of objects and their relations. In its last version, GVT relies on the Unity 3D engine and the distribution of an application developed using GVT is managed by Unity 3D. Moreover, the development of an application through GVT is really constrained, the modules of GVT are inter-dependants.

A more recent initiative, called ARTiFICe [13], proposes a framework to develop both distributed and collaborative applications as well as augmented reality applications. They do not propose a proper interaction design model since they focus more on a solution that can be easily adapted to support novel devices and interaction techniques. Several existing interaction techniques are proposed (VirtualHand[12], Go-Go[14], ...) but they are not customizable. Besides, ARTiFICe relies on Unity 3D making it dependent from this engine. The distribution model is based on the Unity 3D network layer enabling an efficient deployment on several platforms. Moreover, ARTiFICe allows to run simultaneous executions of a VE application, however, collaborative control of a single object is not possible. The strong coupling to Unity makes this solution not independent and not versatile.

It is worth noting that another initiative called FIVE (Framework for Immersive Virtual Environments) has already been proposed in 1997 by Slater et al. [16]. Their work focuses on immersion and presence in VE and the purpose of this framework is to make presence a central concern in VE. Therefore, it does not address the same issues.

2.2 Reflecting on an Easier and Faster Development of CVE

The issues raised by the building of VE applications are pointed by several works as in [20]. In an exhaustive study, the authors identify no less than 67 issues on VE design and implementation, that they arrange in 3 theme categories: human aspect, design and development. To conclude this study, several research challenges are proposed and two of them are close to the #FIVE concerns. First, they claim that representation and functionality must be divided into layers to narrow their focus since no single tool can address every VE needs. Second, they state that models of systems must be proposed:

> *A model operates at a higher level of representation; affording easier iterative development, improving reuse, containing complexity, reducing chaos, and addressing many of the issues with callbacks and events.*

Moreover, in [18], Taylor et al. discussed the lessons learned from 20 years of VE building. The creators of the widely used VRPN library [19] for VR peripheral system discuss several topics concerning VE design and implementation. On the topic related to object interaction, they recommend to split application specific interaction behaviours from the general event behaviour in VE systems. Thus, all sorts of interaction techniques can be built and used generally while the application specific behaviour remains the only part to be implemented on a case-by-case basis.

This is in accordance with the observations shared by Steed in [17] who proposes some abstractions to make VE platforms reusable. In this paper, the author justifies his point of view by the fact that over a 15-years period, at least 40 VE software systems have been used in his lab.

Eventually, in [7], the authors are especially interested in the development of collaborative VE. They highlight three limitations that must be addressed by CVE frameworks. First, CVE frameworks should provide facilities to focus more on the behaviour of the shared virtual objects than on 3D graphics management or distribution and synchronisation issues. Second, they should allow designers to describe the content of the shared world in order to make the shared virtual objects easier to instantiate and configure. Lastly, CVE frameworks should provide support for the deployment on different rendering devices.

2.3 Discussion

None of the frameworks presented in section 2.1 answer all our needs. To our knowledge, there is no existing framework for CVE that covers all our requirements. Yet, these requirements follow the recommendations that can be found in the literature. Indeed, our will to provide an high-level interaction model is in accordance with the statement for systems of model in [20], the statement for splitting interaction behaviour from event behaviour in [18] or the statement for focusing more on the behaviour of the shared virtual objects than on 3D graphics management or distribution and synchronisation issues in [7].

The three other criteria result from the same idea. Independence ensures that the framework is decoupled from hardware or software issues. In the same way, enabling versatility makes it possible to cover more needs and to answer more technical constraints that developers can met. Furthermore, the factorization concern reflects our will to not rebuild from scratch each time we begin the development of a new VR application. Eventually, we observed that, for the past 5 years, powerful game engines have been released. They propose an all-in-one solution to quickly and easily generate game-like real-time VE. A framework for CVE should take advantage of this kind of tool while remaining technologically independent.

3 Overview of #FIVE

#FIVE is a novel framework that offers a set of models to ease the development of new VE applications in order to make them: collaborative, interactive and distributed. First, #FIVE provides a model to ease the declaration of possible actions of objects in the VE. We need, for instance, to easily declare that every screw in the virtual world can be involved in a screwing task. Furthermore, we want to easily declare that a relation exists between a screwdriver and a screw that can result in a screwing action. Second, #FIVE gives a model to facilitate the setting and the management of collaborative interactions in a VE. Indeed, being able to collaboratively control an object is an essential task in CVE, therefore, it is relevant to support an easier set-up of how objects can be controlled and how concurrent control order must be managed. Third, the models proposed by #FIVE are distribution-friendly in order to ease the distribution of the VE for multiple users. Given the variety of available setups and use-cases of CVE, it makes sense to take into account the distribution of a VE early in its development. That is why, #FIVE's inner models is not only compliant with multiple platforms (desktop, CAVE, tablet, ...) but they also provides support for distribution. Fourth, #FIVE provides guidelines that allow users to be more efficient in the development of the VE. In fact, #FIVE can be integrated in a VE in several ways. It lets the developer free of its implementation choices but guidelines are nonetheless proposed based on our experience on CVE design and implementation.

As shown in figure 1, #FIVE is, for now, composed of two core components : the *relation engine* and the *collaborative interaction engine*. The **Relation Engine** computes the possible the actions of objects in the virtual environment whereas the **Interaction Engine** manages the interactions performed by users (collaborative or single-user ones) in the virtual environment. Both of them provide an API dedicated to communication with the VE as well as communication between each other. For instance, an action triggered by the *relation engine* can trigger an interaction that uses the *collaborative interaction engine*.

#FIVE also assists the developer with the distribution of the virtual environment and manages collaborative interaction on the objects of the world.

The main originality of #FIVE is that it lets developers free of their implementation; letting them free to use any engine or third-party tool they feel comfortable with. Indeed, recent years have shown us that the VR domain is subject to lots of changes from hardware devices to rendering and interface techniques as well as softwares. Hence, this strong constraint in the design of #FIVE. Thus, the interactions managed by #FIVE are low-level making it possible to use #FIVE with any kind of input device whatever the degrees of freedom enable by the device. Indeed, #FIVE does not describe interactions but rather proposes a system that can hide the complexity of data exchange for collaborative interactions. We do not work on the semantic level, which is why we do not consider protocols for description but it can be added by the developer.

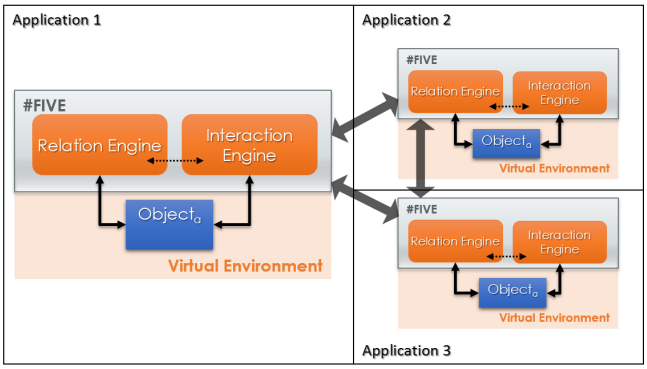

Figure 1: Architecture of #FIVE : collaborative interaction between several instance of the same simulation are supported by #FIVE. #FIVE also supports the definition of action of objects of a VE.

As for the actions and relations of objects, even if #FIVE manages their run-ability, the precise action is described by the developer inside the VE.

Besides, #FIVE does not force the developer to use all the modules. Finally, another benefit of #FIVE is that it makes it possible to reuse components from one application to another. Several components developed for a #FIVE-based application can be reused (see figure 2): the abilities of objects and their behaviours as well as the descriptions of the interactions.

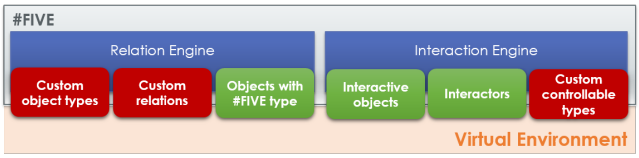

Figure 2: The VE and the #FIVE engines (blue) can use at once custom reusable components (red) and application specific elements (green).

The next two sections present the functioning of the relation engine and of the *collaborative interaction engine* then the section 6 gives examples of the integration of #FIVE in a virtual environment applications.

4 The Relation Engine

The *relation engine* of #FIVE proposes a set of models to support the definition of actions and behaviours of objects of a VE. Its role is to determine how and which objects of the world can be used. In this section, we present the data model behind the *relation engine*. Afterwards, we explain how this data model is used by the *relation engine* and we illustrate this with an example. We conclude this section by a discussion on the benefits of the *relation engine*.

4.1 Data Models of the Relation Engine

The data model of the *relation engine* is based on two concepts: *relational object* and *relation*. First, we call **relational object** any

object of the world that can be involved in a ***relation***. For that purpose, **#FIVE types** are attached to **relational objects**. **#FIVE types** (noted *type* in the remainder of the paper) are properties that an object has and that is required for involving it in a ***relation***. Each *type* of an object has an identifier that is unique in the context of the object. Next, we call a ***relation*** a template of actions; it is defined using a set of *object patterns*. **Object patterns** are patterns of object that are defined by a set of *type*. Figure 3 illustrates these two main concepts: a *relational object* O_a that has two types (t_1 and t_2) and a *relation* R_a that connects two object patterns (X and Y).

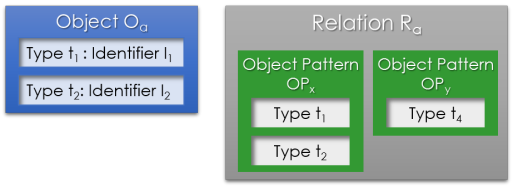

Figure 3: The 2 main concepts used by the *relation engine*. The object O_a has two #FIVE *types* : t_1 and t_2. The *relation* R_a connects two object patterns : one describing objects that possess at once t_1 and t_2 at least, and another for objects that possess t_4 at least.

For instance, let us suppose that we have a threaded rod with two threaded extremities and a nut in our virtual world. In order to involve them in a screwing action ruled by the *relation engine*, we must attribute *type*s to both objects. As the threaded rod (noted O_{Rod}) has two threaded extremities, it can be typed as "male" twice. In the same way, the nut (O_{Nut}) can be typed as "female" twice. We call the two male types of O_{Rod} "$Extremity\ a$" and "$Extremity\ b$" respectively and the two female types of (O_{Nut}) "$Side\ a$" and "$Side\ b$" respectively :

$$O_{Rod} = (t_{male} : Extremity\ a,\ t_{male} : Extremity\ b)$$
$$O_{Nut} = (t_{female} : Side\ a,\ t_{female} : Side\ b)$$

Then, to be able to perform the action "screw" in our virtual environment, we can define a *relation* screw. This *relation* connects two objects; one must be typed as *female* and the other must be types as *male*. We define the *relation* R_{Screw} with two *object patterns* OP_x and OP_y that define this requirements like this :

$$R_{Screw} = OP_x(t_{male}), OP_y(t_{female})$$

Of course, any other object of the world that possesses the types t_{male} and t_{female} can be involved in the relation R_{Screw}. If we want a more specific screwing relation that only takes into account nuts, we must add a type to every nut (t_{nut}) in our world and define a new relation similar to R_{Screw} except that it also contains the *object pattern* that defines nuts like this :

$$R_{Screw-Nut} = OP_x(t_{male}), OP_y(t_{female}, t_{nut})$$

4.2 Usage of the Relation Engine

Based on this data model, the *relation engine* can instantiate *relations*. We call, the instantiation of a *relation* a ***realization***. For a *relation* to be instantiated as a *realization*, each one of its *object pattern* must match a *relational object* of the world. There is a match between a *relational object* O_a and an *object pattern* OP_x when O_a possesses at least all types contained in OP_x. Using our sample of threaded rod and nut, a possible *realization* of the relation R_{Screw} could be instantiated with one extremity of the rod and one extremity of the nut. Thus, a possible *realization* for the *relation* R_{Screw} can be realized with $Extremity\ a$ of O_{Rod} and the $Side\ a$ of O_{Nut} :

$$R_{Screw} = (O_{Rod}(Extremity\ a), O_{Nut}(Side\ a))$$

Indeed $Extremity\ a$ of O_{Rod} matches the first object pattern of R_{Screw} and *side a* of O_{Nut} matches the second object pattern. Another possible *realization* is :

$$Re_{Screw} = (O_{Rod}(Extremity\ a), O_{Nut}(Side\ b))$$

Before executing the *realization*, the actual value of variables that takes part into the action defined by the *realization* must be checked. Indeed, the developer must specify in the *realization* on which conditions it can be run, for example, using the state of the world, the parameters of the objects involved or the state of the object. For that purpose, an intermediate step is runned to ensure that the *realization* can really be runned given data external to the *relation engine*.

In order to help the developer to trigger *realizations* from the context of the VE, the API of the *relation engine* provides methods to trigger *realizations*. To do so, the API provides a *query interface* that proposes methods to interrogate the *relation engine*. The *relation engine* can actually answer several queries deduced from the completion of its data model. This completion results from the declaration of *relational objects* and *relations*. The *relation engine* can answer three basic queries:

1. What are the *realizations* where all objects of a given set of objects can be involved ?
2. What are the possible *realizations* of a given relation ?
3. What are the possible *realizations* given a relation and a set of objects ?

The figure 4 illustrates all three queries. In our sample VE, we have four objects (0_a,0_b,0_c and 0_d) and one *relation* (R_a) declared. Query 1 takes a set of objects (0_a and 0_d) and gives the possible *realizations*. In our case, only one *realization* is possible : $R1$. Query 2 takes a *relation*, (R_a), and gives all possible *realizations*. Here, two *realizations* are possibles : $R1$ and $R2$. Finally, query 3 takes a *relation* (R_a) and a set of objects (0_a and 0_c) to give all possible *realizations*. Given our input data, the only possible *realization* is $R2$.

4.3 Benefits of the Relation Engine

The data model and the capacities of the *relation engine* make it possible to "annotate" a virtual world in a rather straightforward way. Indeed, the concept of *type* is close to the way humans apprehend their surrounding environments; mentally arranging the available objects function of their capacities. Thus, the same *type* can be attributed to several objects if they share the same property. Furthermore, the concept of *type* and *relation* is flexible and extensible: as many *types* as required by the application can be defined. Moreover, *type*s and relations can be re-used in other applications. Thanks to #FIVE, we can not only re-used 3D models but also their behaviours and the actions they may be involved in. Using the standard signature defined in *relation engine*'s API, relations are accordingly hard-coded by the developer, making the link with the VE. Indeed, when simulated by the virtual environment, a nut will always be screw-able in any virtual world and the action of screwing will result in (slightly) the same behavior for this nut.

The usage of the virtual environment is also rather intuitive. Based on our experience in interactive VE, we extract the most useful queries but, if required by a specific application, new ones can easily be added. Besides, the time-response of the queries provided through the API is compatible with real-time VE. The *relation engine* can be interrogated any time during a simulation, taking into account the context at the time of the query. It allows the developer to propose to users a list of every possible actions at a given time function of the context (state of the world, of objects,...). Thus, the

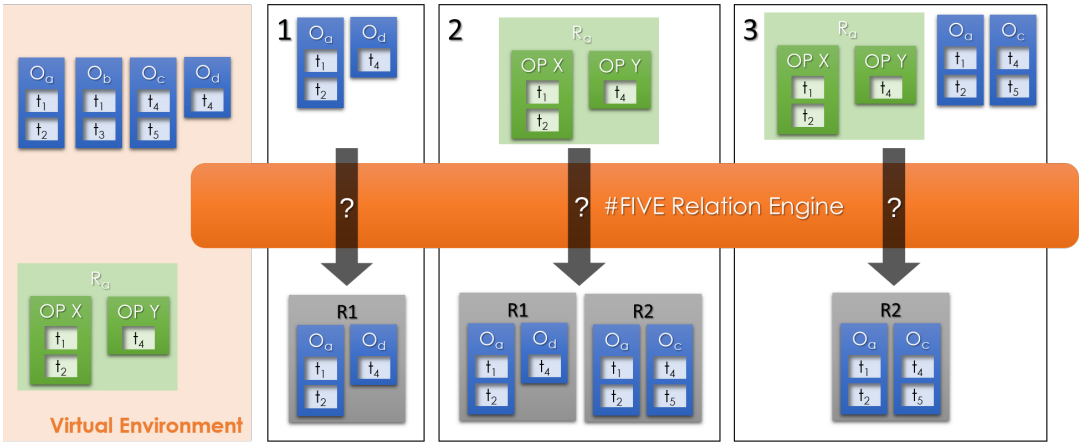

Figure 4: Given the *relational objects* and the *relations* declared for the VE through #FIVE, the *relation engine* can answer 3 types of queries to check what are the possible actions anytime during the simulation runned by the VE.

relation engine opens new perspectives and possibilities of interaction in a VE by providing efficient tools to the developer to be aware of every possible actions continually during a simulation.

5 The Collaborative Interaction Engine

The purpose of the *collaborative interaction engine* of #FIVE is to provide a set of models for the collaborative control of virtual objects. In this section, we first detail the data model behind our *collaborative interaction engine* and, afterwards, we explain its usage and its features. To conclude this section, we discuss the possibilities offered by the *collaborative interaction engine* in a VE.

5.1 Data Models of the Collaborative Interaction Engine

With a view to keeping the model as generic as possible, the data model behind the *collaborative interaction engine* relies on only two distinct entities: *interactive objects* and *interactors* (see figure 5). On the one hand, an **interactive object** can be any object in the virtual world that possesses a set of *controllable parameters*. **Controllable parameters** are parameters that can be modified by an interaction. A controllable parameter is defined by an access manager, a data type, a data identifier and a merger. The access manager defines the access rights that rules the controllable parameter. The **merger** describes the merging policy employed when concurrent modifications are induced by a collaborative interaction.

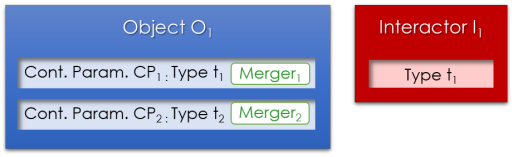

Figure 5: In order to use the *collaborative interaction engine*, objects of the VE are given controllable parameters of different types (0_1 has 2 controllable parameters CP_1 and CP_2). Besides, interactors are defined along with a type that specifies which type of controllable parameters they can modify.

For instance, let us take the nut of the example presented in the previous section. We want the nut to be controlled in two ways: its rotation around the Y axis and its translation on the Y axis. Thus, we can add to the nut two controllable parameters like this :

$$O_{Nut} = (RotationY : Float\ (Merger_1),\ TranslationY : Float\ (Merger_2))$$

On the other hand, an **interactor** is an entity that can alter a controllable parameter. Each *interactor* comes with a type that defines the type of parameter it can control. An *interactor* can be attached to an object if we need to represent it visually in the VE. Besides, an *interactor* describes how the interaction should be performed given the parameter. For instance, to interact with the nut, we can add an *interactor* for rotating an object around an axis. Our *interactor* will only be allowed to control parameters typed as *float*. We note our *interactor* like this :

$$I_{Rotate} = (float)$$

5.2 Usage of the Collaborative Interaction Engine

An interaction managed by the *collaborative interaction engine* is always initiated by an *interactor*. Moreover, an *interactor* is only able to control objects that possess a *controllable parameter* that matches its own type.

Interactions are made up of 5 steps :

1. The *interactor* asks permission to an interactive object to access one of its controllable parameter.
2. If the access is granted then the interaction begins. The *interactor* sends a value to the *controllable parameter*. The way the value is computed is defined in the *interactor* and is specific to an *interactor*. For example, the *interactor* can be linked to an input device that provides it some values. The *interactor* processes it before sending a value to the *controllable parameter* of an object.
3. If the *controllable parameter* is already being accessed by another interactor, the values received are merged according to the merging policy defined in its merger. For example, when controlling the position of an object, if the merging policy is the mean between the values then the position of the object will be halfway between the position asked by the two interactors.
4. The value of the *controllable parameter* is actually updated and the state of the interactive object is updated accordingly.
5. The *interactor* releases its control over the *controllable parameter*.

Figure 6 summarizes the usage of the *collaborative interaction engine* when the virtual environment is distributed on two different platforms with one user on each. As shown in the example 1

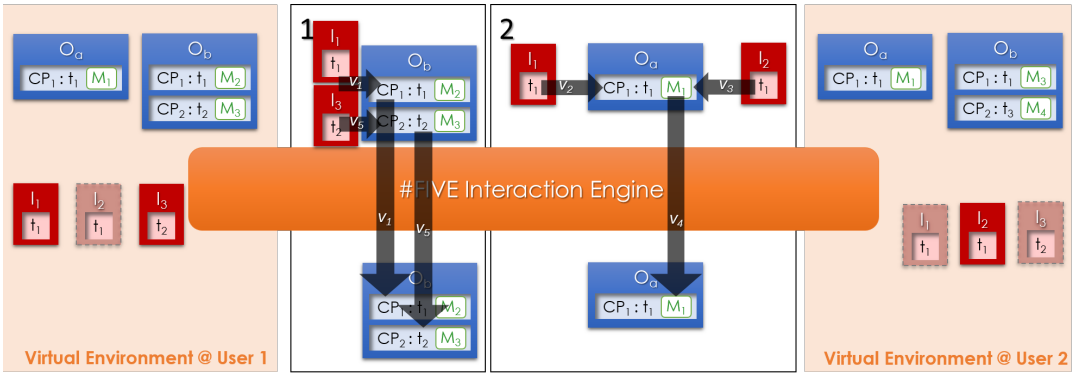

Figure 6: Collaborative interactions with the *collaborative interaction engine*. In the first case, *interactor* I_1 is controlling CP_1 whereas I_3 is controlling CP_2. Their values (respectively (v_1) and (v_5)) are send to the object 0_b. In the second case, two interactors (I_1 and I_2) access CP_1, sending value v_2 and v_3 respectively. A merged value (v_4) is send to O_a according merger M_1.

in figure 6, an *interactor* can only alter one controllable parameter but one object can be controlled by several interactors; each one of them controlling one *controllable parameter*. This is useful to make complex interaction that results in complex trajectories for the controlled object. For instance, if we want to simulate a screwing task with our nut by translating it while it rotates, we must use another *interactor* that will control its translation. The other *interactor* could be noted like this :

$$I_{Translate} = (float)$$

5.3 Benefits of the Collaborative Interaction Engine

The API of the *collaborative interaction engine* provides a set of methods to easily settle how users can collaboratively control objects. The *collaborative interaction engine* not only takes into account access rights and concurrent interactors but can also merge the commands of several *interactors*, using a customizable merging strategy. Furthermore, the model of the *collaborative interaction engine* is independent from what triggers the interaction; it can be any event happening in the VE: proximity, user input as well as a *realization* launched by the *relation engine*. In fact, thanks to this model, the interactions managed by the *collaborative interaction engine* can be independent of the inputs of a specific device, whatever its degrees of freedom. Moreover, by using the model behind the *collaborative interaction engine*, it is possible to chain interactions. An *interactor* I_a can indeed possess controllable parameters that another *interactor* I_b can control. While I_b controls I_a, I_a can control another object. An example of chained interactions can be a hand that controls a screwdriver that itself controls a screw. Furthermore, *interactors* can be reused in several application. Indeed, interactors can describe complex interaction (opening a drawer, screwing, ...) so that it is interesting to reuse it in another virtual environment.

6 Integration of #FIVE in Unity 3D

In this section, we present the integration of #FIVE in the context of Unity 3D. We only rely on Unity 3D for authoring tools and network data transfer so that other platforms can similarly be used such as Simulator X [10]. We first discuss about the adaptation of our framework with the 3D engine and the need of a common interface in-between. Motivations and their corresponding applications are addressed step by step. We finally show several demonstrators as concrete implementations based on this work. They focus on this or that aspect of #FIVE to enforce the understanding. However, as it is still an ongoing work, these applications tend to integrate all the features of the framework.

6.1 The Unity 3D-#FIVE layer

The core modules of the framework are implemented using C#. They are compiled as DLL. They are distributed in this form coming along with their documentation (i.e. mainly a detailed API and developer guidelines).

Our main concern in the integration process was to transform the Unity 3D editor in a #FIVE authoring interface following the already existing UI philosophy of Unity 3D; each #FIVE entity (e.g. *relation*, object, interactor, ...) matches a Component linked to a Game Object of the Scene and it is fully configurable from the Inspector (c.f. Figure 7). This mapping emphasizes the straightforwardness of the model and focuses on the ease-of-manipulation of its concepts.

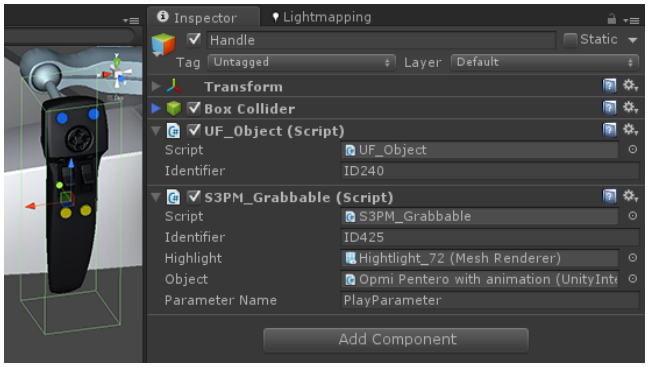

Figure 7: The selected Game Object stands for a relational object which can be involved in a *relation*; it has a UF_Object Component. Its *type* states it has the ability to be grabbed; it has the S3PM_Grabbable Component.

Although this set-up enables authors to script their own #FIVE items from software abstractions, we aimed at providing them essential entities as Unity 3D components off-the-shelf. We are growing this set empirically on the basis of our subsequent implementations of the framework. For instance, we noticed that "take" (i.e. some user can take some object) is a highly common *relation*, likewise "pull" (i.e. a user moves an object along a predefined axis) is a very frequent interaction. This catalogue considerably speeds-up the building of simulations, developers only need to focus their creativity on more specific behaviours.

We were also looking to make #FIVE profit of high level features provided by Unity 3D such as networking. Our integration takes

advantage of the Unity 3D networking API to implement the distribution model of #FIVE. For instance, interactions can be performed in different executions of a same simulation, on different platforms and by different users acting simultaneously. This aspect enhances the collaboration capability of #FIVE and makes it available at no extra-effort for the Unity 3D developers.

Finally, we intended to supply a strong information resource to the 3D designers to value this integration intention. The extension on the top of #FIVE and Unity 3D comes along with a reference documentation. This document is an exhaustive index of the Unity 3D components we implemented, their parameters and their authorized values. However the added-value of this manual lies in the listing of the possible usages of the framework. Although the full potential cannot be addressed, we aim at covering many application as a feedback to the developers. For example, we suggest that the running of a *relation* can trigger an interaction, furthermore a third-party scenario engine can impact the set of allowed *relations*, etc. In addition, these software engineering guidelines are implemented in demonstrators included in the package which comes with the Unity 3D-#FIVE layer.

6.2 The Bolt demonstrator

This minimalist demonstrator consists of a threaded rod with two nuts at its extremities. They can be screwed back and forth along the stem by multiple users at the same time. They can choose to apply concurrent or opposite forces on a same nut to speed-up or to slow-down its movement. Above everything else, this implementation strongly highlights the distributed faculties of the *collaborative interaction engine*. Indeed, simultaneous interactions are echoed in all the executions of the simulation using the networking feature of the Unity 3D-#FIVE layer. Furthermore the merger function of the engine handles the way concurrent interactions are taken into account.

Since there is a unique *relation* defined for this demonstrator (i.e. linking of a user, a threaded rod and a nut), the set-up is very limited and there is no dynamicity in the scenario, the simulation does not really benefit from the query interface of the *relation engine*; the processed *realizations* are quite expected.

Thanks to the multi-platform export feature of Unity 3D, the interoperability of #FIVE has been put to test and confirmed. Using the MiddleVR middleware, this scene was executed concurrently on different hardware systems: Android tablet, iPad, Oculus Rift DK2 with Razer Hydra, VR rooms (a.k.a. CAVE), desktop and laptop computers (c.f. Figure 8).

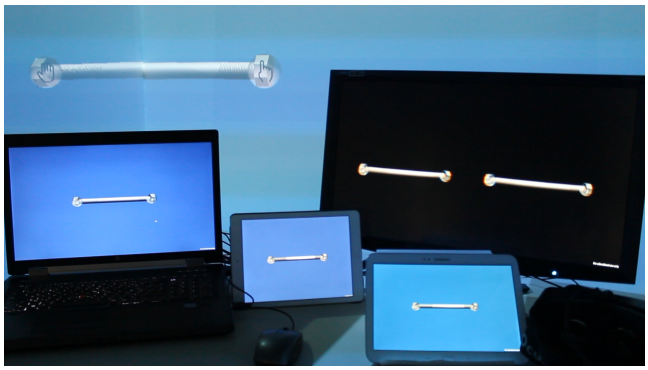

Figure 8: The Bolt demonstrator is running simultaneously on various platforms: a CAVE in the background, a laptop computer on the left, the video capture of an Oculus DK2 on the right, an iPad and an Android tablet in the foreground.

6.3 The Operating Room demonstrator

The ambition of this demonstrator is to train nurses to prepare an operating room prior to a surgery. The trainee is mainly asked to move objects to their target location. A scenario develops the procedure giving visual hints using a colour-based code; take-able surgical instruments are highlighted in blue, handles of the movable apparatus are yellow, scenario targets are green and so on. (c.f. Figure 9).

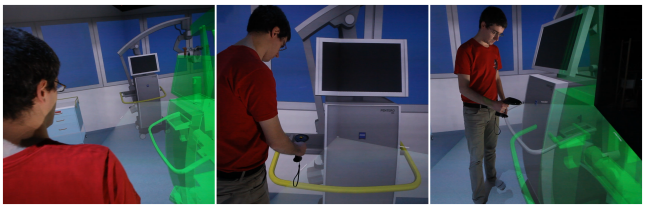

Figure 9: The user must move the appliance to match its green ghost. He can grab it from the yellow handle. Then he can bring it to its target.

The application demonstrates an eventual conjunction of #FIVE with an external scenario module, here, the #SEVEN scenario engine [6]. This narrative engine defines the whole of the *relations* at every key moment of the scenario. Once a *relation* is introduced, the objects of the scene which are potentially involved in it are tinted; in blue whether they contribute to the ”take” *relation*, in yellow if they are involved in ”move”, etc.

This implementation also makes an extensive use of the query interface of the *relation engine*. It is interrogated each time the scenario engine updates the set of the *relations* in order to determine the objects to highlight in the scene. It is questioned anew when the *type* of a relational object is modified. For instance, when the nurse participates to a *realization* she can no longer be involved in any other *relation*.

There are numerous possible interactions in this scene: the user can open and close drawers, push and pull appliances on the floor, open and close doors, etc. As well, this demonstrator is a proof-of-concept which reveals that various interactions can be implemented with very few predefined generic components from our suite: moving along an axis, moving on a plane, rotating along an axis, etc.

6.4 The Bar demonstrator

In this demonstrator, the user finds himself in a bar with many items. Some of them have no real use, they can just be taken and thrown on the ground. Some others can be combined in order to make a tool which can repair the flawed light system of the place.

Building this application was the opportunity to experiment different graphical metaphors to highlight the interactive objects of the scene. Besides the Operating Room demonstrator, this implementation introduces a visual representation of the *relations*: when the user targets an object with a pointing device, it is linked with semi-transparent rays to the other objects which it is related (c.f. Figure 10).

This application also proves that our system can dynamically handle the updates of the world: combination, introduction, destruction and modification of relational objects. In our combination use-case, the query interface of the *relation engine* is used to determine and to recompute the impacted *relations*.

7 Discussion and Conclusion

In this paper, we present #FIVE, a framework that provides a set of models to support the development of collaborative and interactive virtual environments. Its particularity resides in the fact that #FIVE has been conceived with no assumption on the implementation of

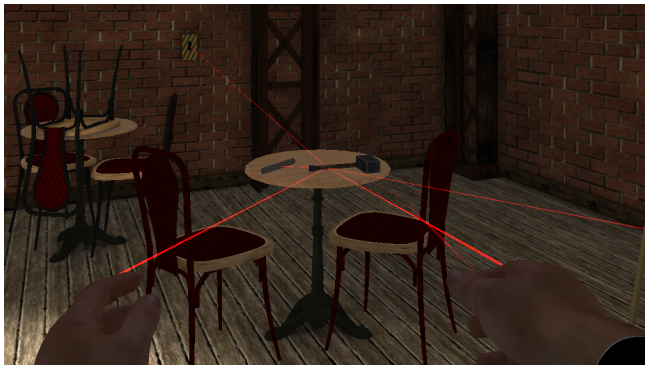

Figure 10: The user points the hammer. The rays show that it can take part in a relation with the right hand and/or the left hand and/or the switch, etc. It gives the user indications of what can be performed.

the final virtual reality application. Indeed, it is not dependent on a specific rendering engine nor on any hardware configuration. In fact, #FIVE gives to the developer, an abstraction layer that support the process of making a VE interactive, collaborative and distributed. Thanks to the two core module of #FIVE, the *relation engine* and the *collaborative interaction engine*, the developer is given a versatile tool to precisely describe the behaviours of the object of the world, the relations between the objects as well as how an object can be manipulated. Besides, the models proposed by #FIVE allows the user to develop customizable components that can be reused from an application to another.

The development of the framework is a work in progress. We are currently looking forward to release a second version. Firstly, this will introduce a user model: their role, their ability, their knowledge, etc. This way, the system will filter the interaction options of real and/or virtual users. Secondly, an all-knowing observer will analyse and evaluate the actions of the users in real time or retrospectively for pedagogical reasons. This will offer to #FIVE a training orientation and make it ready for CVET. Lastly, a porting to another 3D engine is under consideration, in order to demonstrate the independence of the framework regarding virtual environments.

Acknowledgements

This publication is supported by the S3PM project of the Comin-Labs Excellence Center and the FUI PREVIZ.

References

[1] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr: a middleware for large scale virtual reality applications. In *Euro-par 2004 Parallel Processing*, pages 497–505. Springer, 2004.

[2] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: a virtual platform for virtual reality application development. In *IEEE Virtual Reality, 2001. Proceedings*, page 8996, Mar 2001.

[3] C. Blanchard, S. Burgess, Y. Harvill, J. Lanier, A. Lasko, M. Oberman, and M. Teitel. Reality built for two: a virtual reality tool. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 35–36. ACM, 1990.

[4] C. Buche, R. Querrec, P. De Loor, and P. Chevaillier. Mascaret: pedagogical multi-agents systems for virtual environment for training. In *Cyberworlds, 2003. Proceedings. 2003 International Conference on*, pages 423–430. IEEE, 2003.

[5] C. Carlsson and O. Hagsand. DIVE a multi-user virtual reality system. In *Virtual Reality Annual International Symposium, 1993., 1993 IEEE*, pages 394–400. IEEE, 1993.

[6] G. Claude, V. Gouranton, R. Bouville Berthelot, and B. Arnaldi. Short Paper: #SEVEN, a Sensor Effector Based Scenarios Model for Driving Collaborative Virtual Environment. In T. Nojima, D. Reiners, and O. Staadt, editors, *ICAT-EGVE, International Conference on Artificial Reality and Telexistence, Eurographics Symposium on Virtual Environments*, pages 1–4, Bremen, Germany, Dec. 2014.

[7] T. Duval, A. Blouin, and J.-M. Jézéquel. When Model Driven Engineering meets Virtual Reality: Feedback from Application to the Collaviz Framework. In *Software Engineering and Architectures for Realtime Interactive Systems Working Group*, 2014.

[8] S. Gerbaud, N. Mollet, F. Ganier, B. Arnaldi, and J. Tisseau. GVT: a platform to create virtual environments for procedural training. In *Virtual Reality Conference, 2008. VR'08. IEEE*, pages 225–232. IEEE, 2008.

[9] G. D. Kessler, D. A. Bowman, and L. F. Hodges. The simple virtual environment library: an extensible framework for building VE applications. *Presence: Teleoperators and virtual environments*, 9(2):187–208, 2000.

[10] M. E. Latoschik and H. Tramberend. Simulator x: A scalable and concurrent architecture for intelligent realtime interactive systems. In *Virtual Reality Conference (VR), 2011 IEEE*, pages 171–174. IEEE, 2011.

[11] J.-L. Lugrin and M. Cavazza. Making sense of virtual environments: action representation, grounding and common sense. In *Proceedings of the 12th international conference on Intelligent user interfaces*, pages 225–234. ACM, 2007.

[12] M. R. Mine, F. P. Brooks Jr, and C. H. Sequin. Moving objects in space: exploiting proprioception in virtual-environment interaction. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 19–26. ACM Press/Addison-Wesley Publishing Co., 1997.

[13] A. Mossel, C. Schönauer, G. Gerstweiler, and H. Kaufmann. Artifice-augmented reality framework for distributed collaboration. *International Journal of Virtual Reality*, 2013.

[14] I. Poupyrev, M. Billinghurst, S. Weghorst, and T. Ichikawa. The go-go interaction technique: non-linear mapping for direct manipulation in vr. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 79–80. ACM, 1996.

[15] C. Shaw, J. Liang, M. Green, and Y. Sun. The decoupled simulation model for virtual reality systems. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–328. ACM, 1992.

[16] M. Slater and S. Wilbur. A framework for immersive virtual environments (FIVE): Speculations on the role of presence in virtual environments. *Presence: Teleoperators and virtual environments*, 6(6):603–616, 1997.

[17] A. Steed. Some useful abstractions for re-usable virtual environment platforms. *Software Engineering and Architectures for Realtime Interactive Systems-SEARIS*, 2008.

[18] R. M. Taylor, J. Jerald, C. VanderKnyff, J. Wendt, D. Borland, D. Marshburn, W. R. Sherman, and M. C. Whitton. Lessons about virtual environment software systems from 20 years of VE building. *Presence: Teleoperators and Virtual Environments*, 19(2):162–178, 2010.

[19] R. M. Taylor II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. Vrpn: a device-independent, network-transparent vr peripheral system. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61. ACM, 2001.

[20] C. Wingrave and J. LaViola. Reflecting on the design and implementation issues of virtual environments. *Presence*, 19(2):179195, 2010.